Software Testing and Validation – 2017/18

Instituto Superior Técnico

Prof. João Dias Pereira

Vos
Project Report

Group 01 – Alameda

Francisca Cambra
ist181057

Rui Ventura
ist181045

April 13, 2018

Contents

1 Method-Scope Tests

1.1 `assignPhoneNumber`

Assigns a free phone number to a client of *Vos* if all conditions are met. If at least one does not hold, then this method does not change anything, throwing an `InvalidOperationException`.

1.1.1 Test Pattern – Category-Partition Test

1.1.2 Functions

- Primary function
  - Assign free phone number to a client without a number
- Secondary functions
  - Throw `InvalidOperationException` if conditions aren't met
    - Invalid nif (nif $\notin [10^8, 10^9[$)
    - A *Vos* client with the given nif doesn't exist
    - Invalid phone number (number $\notin [10^8, 10^9[$)
    - It isn't a *Vos* number
    - Phone number already assigned
    - Client can't be assigned any more numbers

1.1.3 Input/Output Parameters

- Input
  - `clientNif` – The nif of the client to assign a number to
  - `phoneNumber` – The phone number to be assigned
  - `clients` – The set of *Vos* clients managed by `ClientManager`
- Output
  - `client` – The updated client, if a number was assigned successfully

1.1.4 Categories & Choices

Parameter	Category	Choices
`clientNif`	*Vos* client (w/ $\#numbers$ phone numbers)	$\#numbers \in [1, 5[$
		$\#numbers = 5$ (MAX)
	Not a *Vos* client	`clientNif` $\in [10^8, 10^9[$
	Invalid nif	`clientNif` $\notin [10^8, 10^9[$
`phoneNumber`	*Vos* phone number	Free (Unassigned)
		Not free (Assigned)
	Not a *Vos* number	`phoneNumber` $\in [10^8, 10^9[$
	Invalid number	`phoneNumber` $\notin [10^8, 10^9[$
`clients`	n-elements	$n = 0$ (Empty)
		$n \in [1, \text{MAX}]$ (Not empty)

Table 1: Set of `assignPhoneNumber`'s input parameters broken into categories, accompanied by test case choices

1.1.5 Constraints

- Empty `clients` list precludes the assignment of a `phoneNumber` to a client (which, since the list is empty, mustn't exist)
- Assigning an invalid `phoneNumber`, one that doesn't belong to *Vos* or one that is already assigned is the same for any kind of client

1.1.6 Test Cases

	Choices			**Expected Result**	
TC	`clientNif`	`phoneNumber`	`clients`	**Exception**	`client`
1	$\#numbers \in [1, 5[$	Free	$n \in [1, \text{MAX}]$	NO	$\#numbers \in]1, 5]$
2	$\#numbers \in [1, 5[$	Not free	$n \in [1, \text{MAX}]$	YES	—
3	$\#numbers \in [1, 5[$	$\in [10^8, 10^9[$	$n \in [1, \text{MAX}]$	YES	—
4	$\#numbers \in [1, 5[$	$\notin [10^8, 10^9[$	$n \in [1, \text{MAX}]$	YES	—
5	$\#numbers = 5$	Free	$n \in [1, \text{MAX}]$	YES	—
6	$\in [10^8, 10^9[$	Free	$n \in [1, \text{MAX}]$	YES	—
7	$\notin [10^8, 10^9[$	Free	$n \in [1, \text{MAX}]$	YES	—

Table 2: Set of reduced test cases for the `assignPhoneNumber` method after constraints are applied

1.2 `computeBill` method

The responsibility of `computeBill` method is to determine the value to pay for a client taking into account all communications made by the client through all of his registered mobile phones

1.2.1 Test Pattern – Combinational Function Test

1.2.2 Decision Tree

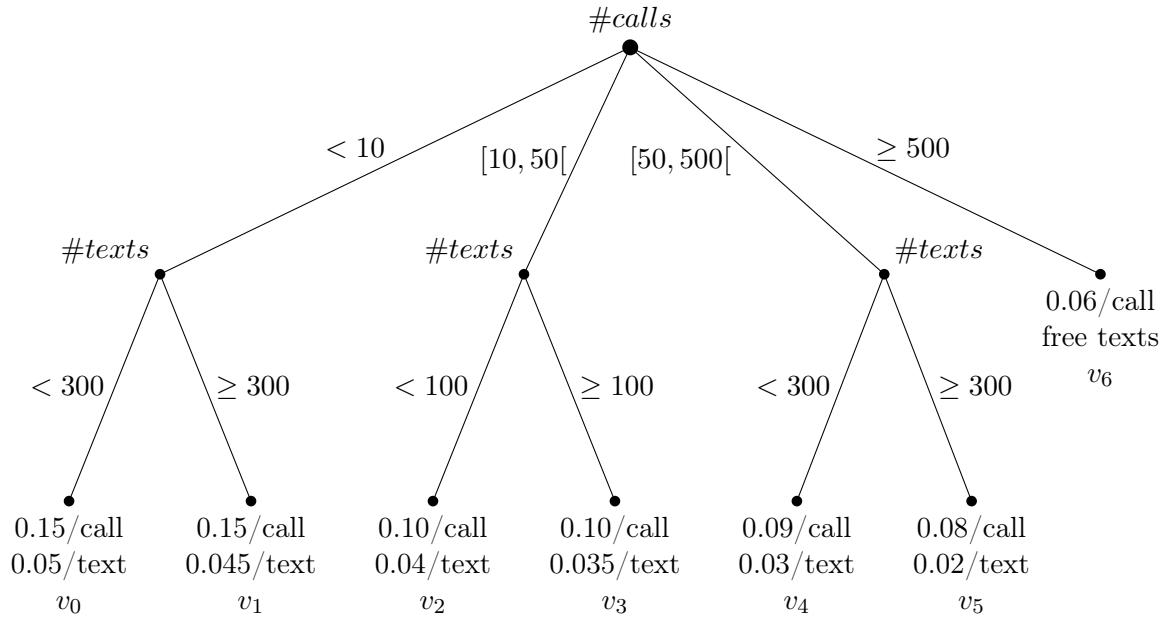

Figure 1: Decision tree describing the output given by `computeBill` based on the number of texts sent and calls made by the client

1.2.3 Domain Matrices

v_0			**Test Cases**			
Variable	**Condition**	**Type**	—	1	—	2
#calls	< 10	ON	10			
		OFF		9		
	Typical	IN			8	7
#texts	< 300	ON			300	
		OFF				299
	Typical	IN	147	204		
Expected Result			v_3	11.55	v_1	16.00

Table 3: v_0 domain matrix

v_1			Test Cases			
Variable	**Condition**	**Type**	—	3	4	—
#calls	< 10	ON	10			
		OFF		9		
	Typical	IN			6	5
#texts	≥ 300	ON			300	
		OFF				299
	Typical	IN	320	400		
Expected Result			v_3	19.35	14.40	v_0

Table 4: v_1 domain matrix

v_2			Test Cases					
Variable	**Condition**	**Type**	5	—	—	6	—	7
#calls	≥ 10	ON	10					
		OFF		9				
	< 50	ON			50			
		OFF				49		
	Typical	IN					22	35
#texts	< 100	ON					100	
		OFF						99
	Typical	IN	48	20	33	15		
Expected Result			2.92	v_0	v_4	5.50	v_3	7.46

Table 5: v_2 domain matrix

v_3			Test Cases					
Variable	**Condition**	**Type**	8	—	—	9	10	—
#calls	≥ 10	ON	10					
		OFF		9				
	< 50	ON			50			
		OFF				49		
	Typical	IN					12	44
#texts	≥ 100	ON					100	
		OFF						99
	Typical	IN	148	220	333	414		
Expected Result			6.18	v_0	v_5	15.49	4.70	v_2

Table 6: v_3 domain matrix

v_4			**Test Cases**					
Variable	**Condition**	**Type**	11	—	—	12	—	13
#calls	≥ 50	ON	50					
		OFF		49				
	< 500	ON			500			
		OFF				499		
	Typical	IN					142	51
#texts	< 300	ON					300	
		OFF						299
	Typical	IN	240	189	98	10		
Expected Result			11.70	v_3	v_6	45.21	v_5	13.56

Table 7: v_4 domain matrix

v_5			**Test Cases**					
Variable	**Condition**	**Type**	14	—	—	15	16	—
#calls	≥ 50	ON	50					
		OFF		49				
	< 500	ON			500			
		OFF				499		
	Typical	IN					200	60
#texts	≥ 300	ON					300	
		OFF						299
	Typical	IN	314	500	616	404		
Expected Result			10.28	v_3	v_6	48.00	22.00	v_4

Table 8: v_5 domain matrix

v_6			**Test Cases**	
Variable	**Condition**	**Type**	17	—
#calls	≥ 500	ON	500	
		OFF		499
Expected Result			30.00	v_4/v_5

Table 9: v_6 domain matrix

2 Class-Scope Tests

2.1 `Client` class

Each client of *Vos* has a name (with a minimal length of 5) and by its social security number (designated as nif). This number is a unique identifier in *Vos*. A client can have several phone numbers managed by *Vos* (between 1 and 5). Each client can associate a mobile phone to each of his assigned phone numbers.

Each client can register in the system a given amount of phone number of *friends*. The maximum number of phone number a client can register is equal to three times the number of phone numbers plus five.

2.1.1 Test Pattern – Non-Modal Class Test

2.1.2 Class Invariant

Client variables	
Variable	**Type**
`name`	`String`
`nif`	`int`
`numbers`	`List<Integer>`
`friends`	`List<Integer>`

Table 10: `Client` class' variables and their respective types

Domain restrictions

- `name.length()` ≥ 5
- `nif` $\in [10^8, 10^9[$
- `numbers.size()` $\in [1, 5]$
- `friends.size()` $\leq 3 \times$ `numbers.size()` $+ 5$

The logical conjunction of all of these restrictions makes up the Class Invariant

2.1.3 On and Off points

Boundary	**ON**	**OFF**
`name.length()` ≥ 5	5	4
`nif` $\geq 10^8$	10^8	$10^8 - 1$
`nif` $< 10^9$	10^9	$10^9 - 1$
`numbers.size()` ≥ 1	1	0
`numbers.size()` < 5	5	4
`friends.size()` $\leq 3n$[1] $+ 5$	$3n + 5$	$3n + 6$

Table 11: On and Off points for the `Client` class' invariant boundaries

[1] `numbers.size()`

2.1.4 Domain Matrix

Boundary			Test Cases											
Variable	**Condition**	**Type**	1	2	3	4	5	6	7	8	9	10	11	12
`name.length()`	≥ 5	ON	5											
		OFF		4										
	Typical	IN			6	7	8	9	10	11	12	13	14	15
`nif`	$\geq 10^8$	ON			10^8									
		OFF				10^8-1								
	$< 10^9$	ON					10^9							
		OFF						10^9-1						
	Typical	IN	10^8+1	10^8+2					10^8+3	10^8+4	10^8+5	10^8+6	10^8+7	10^8+8
`numbers.size()`	≥ 1	ON							1					
		OFF								0				
	< 5	ON									5			
		OFF										4		
	Typical	IN	2	3	4	3	2	3					4	3
`friends.size()`	$\leq 3n+5$	ON											17	
		OFF												15
	Typical	IN	0	1	2	3	4	5	6	7	8	9		
Expected Result			✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✓	✗

Table 12: `Client` class test cases

2.2 `Mobile` class

A mobile phone can make and receive calls and send and receive texts. A mobile phone can be turned on or off (and in this case it cannot make calls, send texts and receive calls nor texts). It has two modes (*friend* and *silent*) that can be enabled or disabled.

2.2.1 Test Pattern – Modal Class Test

2.2.2 Finite State Machine

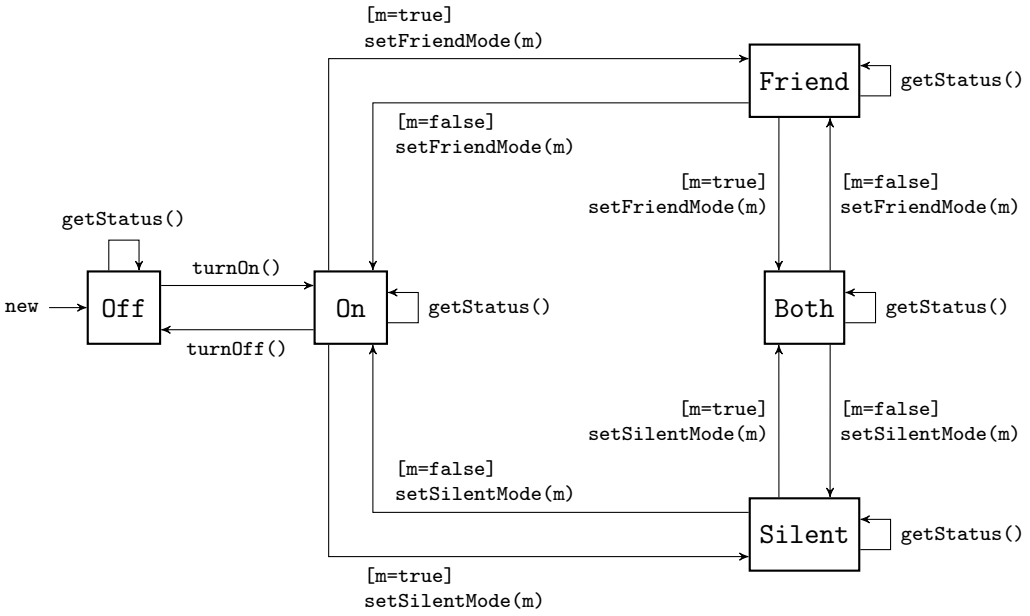

Figure 2: `Mobile` class state machine, representing the class' states and transitions between them

2.2.3 Transition Tree

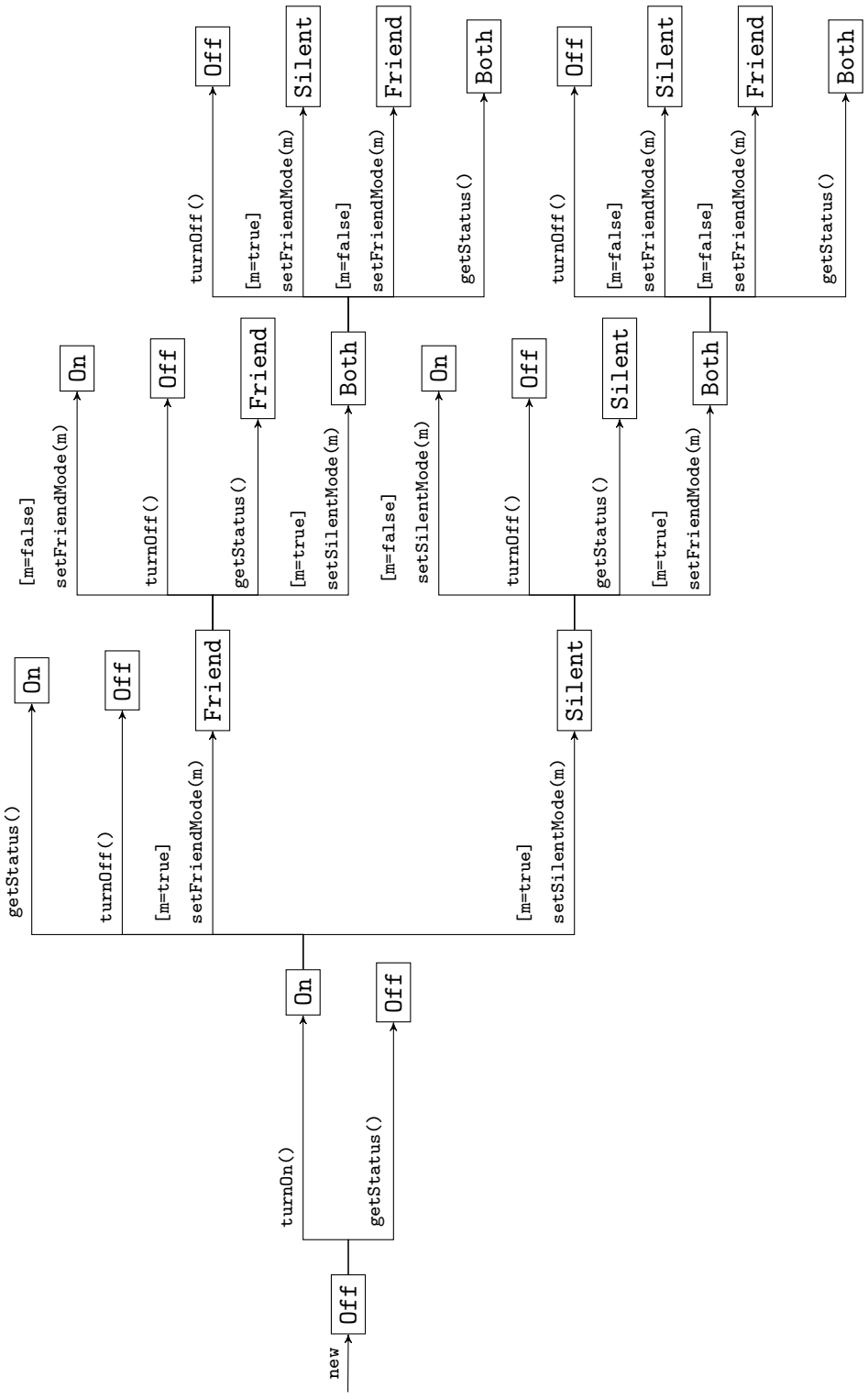

Figure 3: `Mobile` class transition tree. Sneak paths are not represented

2.2.4 Conformance Test Suite

TC	Level 1	Level 2	Level 3	Level 4	Level 5	Next State	Exception
1	new	—	—	—	—	Off	NO
2	new	turnOn()	—	—	—	On	NO
3	new	getStatus()	—	—	—	Off	NO
4	new	turnOn()	getStatus()	—	—	On	NO
5	new	turnOn()	turnOff()	—	—	Off	NO
6	new	turnOn()	[m=true] setFriendMode(m)	—	—	Friend	NO
7	new	turnOn()	[m=true] setFriendMode(m)	[m=false] setFriendMode(m)	—	On	NO
8	new	turnOn()	[m=true] setFriendMode(m)	turnOff()	—	Off	NO
9	new	turnOn()	[m=true] setFriendMode(m)	getStatus()	—	Friend	NO
10	new	turnOn()	[m=true] setFriendMode(m)	[m=true] setSilentMode(m)	—	Both	NO
11	new	turnOn()	[m=true] setFriendMode(m)	[m=true] setSilentMode(m)	turnOff()	Off	NO
12	new	turnOn()	[m=true] setFriendMode(m)	[m=true] setSilentMode(m)	[m=false] setFriendMode(m)	Silent	NO
13	new	turnOn()	[m=true] setFriendMode(m)	[m=true] setSilentMode(m)	[m=false] setSilentMode(m)	Friend	NO
14	new	turnOn()	[m=true] setFriendMode(m)	[m=true] setSilentMode(m)	getStatus()	Both	NO
15	new	turnOn()	[m=true] setSilentMode(m)	—	—	Silent	NO
16	new	turnOn()	[m=true] setSilentMode(m)	[m=false] setSilentMode(m)	—	On	NO
17	new	turnOn()	[m=true] setSilentMode(m)	turnOff()	—	Off	NO
18	new	turnOn()	[m=true] setSilentMode(m)	getStatus()	—	Silent	NO
19	new	turnOn()	[m=true] setSilentMode(m)	[m=true] setFriendMode(m)	—	Both	NO
20	new	turnOn()	[m=true] setSilentMode(m)	[m=true] setFriendMode(m)	turnOff()	Off	NO
21	new	turnOn()	[m=true] setSilentMode(m)	[m=true] setFriendMode(m)	[m=false] setFriendMode(m)	Silent	NO
22	new	turnOn()	[m=true] setSilentMode(m)	[m=true] setFriendMode(m)	[m=false] setSilentMode(m)	Friend	NO
23	new	turnOn()	[m=true] setSilentMode(m)	[m=true] setFriendMode(m)	getStatus()	Both	NO

Table 13: `Mobile` class conformance test suite

2.2.5 Sneak Paths

Events	States				
	Off	On	Friend	Silent	Both
turnOn()	✓	PSP	PSP	PSP	PSP
turnOff()	PSP	✓	✓	✓	✓
setSilentMode(boolean)	PSP	?	?	?	?
setFriendMode(boolean)	PSP	?	?	?	?
sendSMS()	PSP	✓	✓	✓	✓
receiveSMS()	PSP	✓	?	✓	?
makeCall()	PSP	✓	✓	PSP	PSP
acceptCall()	PSP	✓	?	PSP	PSP
getStatus()	✓	✓	✓	✓	✓

Table 14: Possible situations in which a message should be rejected in the given `Mobile` class' state

2.2.6 Sneak Path Test Suite

TC	Level 1	Level 2	Level 3	Level 4	Level 5	Exp State	Exception
1	new	turnOff()	—	—	—	Off	YES
2	new	setSilentMode()	—	—	—	Off	YES
3	new	setFriendMode()	—	—	—	Off	YES
4	new	sendSMS()	—	—	—	Off	YES
5	new	receiveSMS()	—	—	—	Off	YES
6	new	makeCall()	—	—	—	Off	YES
7	new	acceptCall()	—	—	—	Off	YES
8	new	turnOn()	turnOn()	—	—	On	YES
9	new	turnOn()	[m=true] setFriendMode(m)	turnOn()	—	Friend	YES
10	new	turnOn()	[m=true] setSilentMode(m)	turnOn()	—	Silent	YES
11	new	turnOn()	[m=true] setSilentMode(m)	makeCall()	—	Silent	YES
12	new	turnOn()	[m=true] setSilentMode(m)	acceptCall()	—	Silent	YES
13	new	turnOn()	[m=true] setSilentMode(m)	[m=true] setFriendMode(m)	turnOn()	Both	YES
14	new	turnOn()	[m=true] setSilentMode(m)	[m=true] setFriendMode(m)	makeCall()	Both	YES
15	new	turnOn()	[m=true] setSilentMode(m)	[m=true] setFriendMode(m)	acceptCall()	Both	YES

Table 15: Set of test cases able to detect possible sneak paths in the `Mobile` class

3 TestNG Test Cases

3.1 ClientClassTest

```
37  public class ClientClassTest {
38
39    private static final String NAME_ON = "Kevin";
40    private static final int NAME_ON_LENGTH = 5;
41    private static final String NAME_OFF = "Mike";
42    private static final int NAME_OFF_LENGTH = 4;
43
44    private static final int NIF_LO_OFF = 100000000 - 1;
45    private static final int NIF_LO_ON = 100000000;
46    private static final int NIF_HI_OFF = 1000000000 - 1;
47    private static final int NIF_HI_ON = 1000000000;
48
49    private static final Integer[] PHONE_NUMBERS = new Integer[5];
50
51    @BeforeMethod
52    public void setUp() {
53      for (int i = 0; i < PHONE_NUMBERS.length; i++) {
54        PHONE_NUMBERS[i] = 910000000 + new Random().nextInt(90000000);
55      }
56    }
57
58    @Test(testName = "TC01 -- name.length() == " + NAME_ON_LENGTH + " -- ctor")
59    void testCase1a() throws Exception {
60      Client client = new Client(NAME_ON, 100000001, PHONE_NUMBERS[0]);
61
62      assertEquals(client.getName(), NAME_ON);
63      assertEquals(client.getNif(), 100000001);
64      assertEquals(client.getPhoneNumbers(), Collections.singletonList(PHONE_NUMBERS[0]));
65      assertEquals(client.getMobiles(), Collections.emptyList());
66      assertEquals(client.getFriends(), Collections.emptyList());
67    }
68
69    @Test(testName = "TC01 -- name.length() == " + NAME_ON_LENGTH + " -- setName")
70    void testCase1b() throws Exception {
71      Client client = new Client("Angela", 100000001, PHONE_NUMBERS[0]);
72
73      client.addPhoneNumber(PHONE_NUMBERS[1]);
74      client.setName(NAME_ON);
75      assertEquals(client.getNif(), 100000001);
76      assertEquals(client.getName(), NAME_ON);
77      assertEquals(client.getPhoneNumbers().toArray(), Arrays.copyOfRange(PHONE_NUMBERS, 0, 2));
78      assertEquals(client.getMobiles(), Collections.emptyList());
79      assertEquals(client.getFriends(), Collections.emptyList());
80    }
81
82    @Test(testName = "TC02 -- name.length() == " + NAME_OFF_LENGTH + " -- ctor",
83          expectedExceptions = InvalidOperationException.class)
84    void testCase2a() throws Exception {
85      new Client(NAME_OFF, 100000002, PHONE_NUMBERS[0]);
86    }
87
88    @Test(testName = "TC02 -- name.length() == " + NAME_OFF_LENGTH + " -- setName")
89    void testCase2b() throws Exception {
90      Client client = new Client("Watson", 100000002, PHONE_NUMBERS[0]);
91      int friend = 900000000 + new Random().nextInt(100000000);
92
93      for (int i = 0; i < 2; i++) {
94        client.addPhoneNumber(PHONE_NUMBERS[i + 1]);
95      }
96      client.addFriend(friend);
97
98      try {
99        client.setName(NAME_OFF);
100       fail();
101     } catch (InvalidOperationException ioe) {
102       assertEquals(client.getName(), "Watson");
103       assertEquals(client.getNif(), 100000002);
104       assertEquals(client.getMobiles(), Collections.emptyList());
105       assertEquals(client.getPhoneNumbers().toArray(), Arrays.copyOfRange(PHONE_NUMBERS, 0, 3));
```

```
106         assertEquals(client.getFriends(), Collections.singletonList(friend));
107       }
108     }
109
110     @Test(testName = "TC03 -- nif == " + NIF_LO_ON + " -- ctor")
111     void testCase3a() throws Exception {
112       Client client = new Client("Pamela", NIF_LO_ON, PHONE_NUMBERS[0]);
113
114       assertEquals(client.getNif(), NIF_LO_ON);
115       assertEquals(client.getMobiles(), Collections.emptyList());
116       assertEquals(client.getFriends(), Collections.emptyList());
117       assertEquals(client.getName(), "Pamela");
118       assertEquals(client.getPhoneNumbers(), Collections.singletonList(PHONE_NUMBERS[0]));
119     }
120
121     @Test(testName = "TC03 -- nif == " + NIF_LO_ON + " -- setNif")
122     public void testCase3b() throws Exception {
123       Client client = new Client("Pamela", 123456789, PHONE_NUMBERS[0]);
124       List<Integer> friends = Arrays.stream(new Integer[2])
125           .map(f -> 900000000 + new Random().nextInt(100000000))
126           .collect(Collectors.toList());
127
128       for (int i = 0; i < 3; i++) {
129         client.addPhoneNumber(PHONE_NUMBERS[i + 1]);
130       }
131       for (Integer friend : friends) {
132         client.addFriend(friend);
133       }
134
135       client.setNif(NIF_LO_ON);
136       assertEquals(client.getNif(), NIF_LO_ON);
137       assertEquals(client.getFriends(), friends);
138       assertEquals(client.getName(), "Pamela");
139       assertEquals(client.getMobiles(), Collections.emptyList());
140       assertEquals(client.getPhoneNumbers().toArray(), Arrays.copyOfRange(PHONE_NUMBERS, 0, 4));
141     }
142
143     @Test(testName = "TC04 -- nif == " + NIF_LO_OFF + " -- ctor",
144           expectedExceptions = InvalidOperationException.class)
145     void testCase4a() throws Exception {
146       new Client("Schrute", NIF_LO_OFF, PHONE_NUMBERS[0]);
147     }
148
149     @Test(testName = "TC04 -- nif == " + NIF_LO_OFF + " -- setNif")
150     void testCase4b() throws Exception {
151       Client client = new Client("Schrute", 123456789, PHONE_NUMBERS[0]);
152       List<Integer> friends = Arrays.stream(new Integer[3])
153           .map(f -> 900000000 + new Random().nextInt(100000000))
154           .collect(Collectors.toList());
155
156       for (int i = 0; i < 2; i++) {
157         client.addPhoneNumber(PHONE_NUMBERS[i + 1]);
158       }
159       for (Integer friend : friends) {
160         client.addFriend(friend);
161       }
162
163       try {
164         client.setNif(NIF_LO_OFF);
165         fail();
166       } catch (InvalidOperationException ioe) {
167         assertEquals(client.getMobiles(), Collections.emptyList());
168         assertEquals(client.getName(), "Schrute");
169         assertEquals(client.getNif(), 123456789);
170         assertEquals(client.getPhoneNumbers().toArray(), Arrays.copyOfRange(PHONE_NUMBERS, 0, 3));
171         assertEquals(client.getFriends(), friends);
172       }
173     }
174
175     @Test(testName = "TC05 -- nif == " + NIF_HI_ON + " -- ctor",
176           expectedExceptions = InvalidOperationException.class)
177     void testCase5a() throws Exception {
178       new Client("Mendeley", NIF_HI_ON, PHONE_NUMBERS[0]);
```

```
179     }
180
181     @Test(testName = "TC05 -- nif == " + NIF_HI_ON + " -- setNif")
182     void testCase5b() throws Exception {
183       Client client = new Client("Mendeley", 132659847, PHONE_NUMBERS[0]);
184       List<Integer> friends = Arrays.stream(new Integer[4])
185           .map(f -> 900000000 + new Random().nextInt(100000000))
186           .collect(Collectors.toList());
187
188       client.addPhoneNumber(PHONE_NUMBERS[1]);
189       for (Integer friend : friends) {
190         client.addFriend(friend);
191       }
192
193       try {
194         client.setNif(NIF_HI_ON);
195         fail();
196       } catch (InvalidOperationException ioe) {
197         assertEquals(client.getMobiles(), Collections.emptyList());
198         assertEquals(client.getName(), "Mendeley");
199         assertEquals(client.getFriends(), friends);
200         assertEquals(client.getPhoneNumbers().toArray(), Arrays.copyOfRange(PHONE_NUMBERS, 0, 2));
201         assertEquals(client.getNif(), 132659847);
202       }
203     }
204
205     @Test(testName = "TC06 -- nif == " + NIF_HI_OFF + " -- ctor")
206     void testCase6a() throws Exception {
207       Client client = new Client("Charlotte", NIF_LO_ON, PHONE_NUMBERS[0]);
208
209       assertEquals(client.getFriends(), Collections.emptyList());
210       assertEquals(client.getName(), "Charlotte");
211       assertEquals(client.getMobiles(), Collections.emptyList());
212       assertEquals(client.getNif(), NIF_LO_ON);
213       assertEquals(client.getPhoneNumbers(), Collections.singletonList(PHONE_NUMBERS[0]));
214     }
215
216     @Test(testName = "TC06 -- nif == " + NIF_HI_OFF + " -- setNif")
217     public void testCase6b() throws Exception {
218       Client client = new Client("Charlotte", 147258369, PHONE_NUMBERS[0]);
219       List<Integer> friends = Arrays.stream(new Integer[5])
220           .map(f -> 900000000 + new Random().nextInt(100000000))
221           .collect(Collectors.toList());
222
223       for (int i = 0; i < 2; i++) {
224         client.addPhoneNumber(PHONE_NUMBERS[i + 1]);
225       }
226       for (Integer friend : friends) {
227         client.addFriend(friend);
228       }
229
230       client.setNif(NIF_HI_OFF);
231       assertEquals(client.getPhoneNumbers().toArray(), Arrays.copyOfRange(PHONE_NUMBERS, 0, 3));
232       assertEquals(client.getFriends(), friends);
233       assertEquals(client.getNif(), NIF_HI_OFF);
234       assertEquals(client.getMobiles(), Collections.emptyList());
235       assertEquals(client.getName(), "Charlotte");
236     }
237
238   }
```